# SL2002
# Software Design & Architecture Lab
# Lab#13

# JavaDocs and JUnit Testing

## JavaDocs

### Introduction

Have you ever had an experience of reading your code months after you last worked on it? You probably had problems remembering exactly what the code does. We can actually do two things to avoid such a situation. One is to strive to make your program simple and readable. The other is to write good documentation.

Javadoc is a convenient, standard way to document your Java code. Javadoc is actually a special format of comments. There are some utilities that read the comments, and then generate HTML document based on the comments. HTML files give us the convenience of hyperlinks from one document to another. Most class libraries, both commercial and open source, provide Javadoc documents.

Note: Most frequently used tags can be generated by the code template, and you can find others with Content Assist by pressing Ctrl-SPACE.

### Types of Javadoc

There are two kinds of Javadoc comments: class-level comments, and member-level comments. Class-level comments provide the description of the classes, and member-level comments describe the purposes of the members. Both types of comments start with /** and end with */. For example, this is a Javadoc comment:

```
/** This is a Javadoc comment */
```

#### 1. Class-level Comments

Class-level comments provide a description of the class, and they are placed right above the code that declares the class. Class-level comments generally contain author tags, and a description of the class. An example class-level comment is below:

```
/**
 * @author waqas
 * @version 1.0
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system.  The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
```

```
public class Inventory {

   //Inventory code here

}
```

### 2. Member-level Comments

Member-level comments describe the fields, methods, and constructors. Method and constructor comments may contain tags that describe the parameters of the method. Method comments may also contain return tags. An example of these member-level comments are below:

```
/**
 * @author waqas
 *
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system. The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
public class Inventory {

   /**
    * Inventory for coffee
    */
   private int coffee;

   /**
    * Default constructor for Inventory
    * Sets all ingredients to 15 units
    */
   public Inventory() {
      this.coffee = 15;
   }

   /**
    * Returns the units of coffee in the Inventory
    *
    * @return int
    */
   public int getCoffee() {
      return coffee;
   }

   /**
    * Sets the units of coffee in the Inventory
    *
    * @param int new units coffee
    */
   public void setCoffee(int newCoffee) {
      this.coffee = newCoffee;
   }
}
```

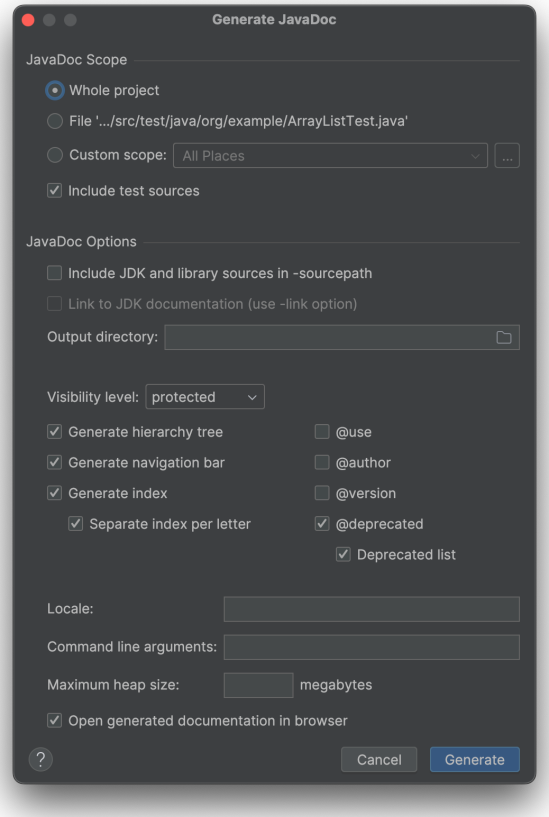
## Tags

***Tags*** are keywords recognized by Javadoc which define the type of information that follows. Tags come after the description (separated by a new line). Here are some common pre-defined tags:

- ***@author [author name]*** - identifies author(s) of a class or interface.
- ***@version [version]*** - version info of a class or interface.
- ***@param [argument name] [argument description]*** - describes an argument of method or constructor.
- ***@return [description of return]*** - describes data returned by method (unnecessary for constructors and void methods).
- ***@exception [exception thrown] [exception description]*** - describes exception thrown by method.
- ***@throws [exception thrown] [exception description]*** - same as ***@exception***.

## Generate HTML Document in IntelliJ

- From the menu bar select **Tools** and then click **Generate JavaDoc.**
- Select the location to store your JavaDocs and click **Generate**.



# JUnit Testing

JUnit is a simple open-source Java testing framework used to write and run repeatable automated tests. Unit testing is the process of examining a small "unit" of software (usually a single class) to verify that it meets its expectations or specification.

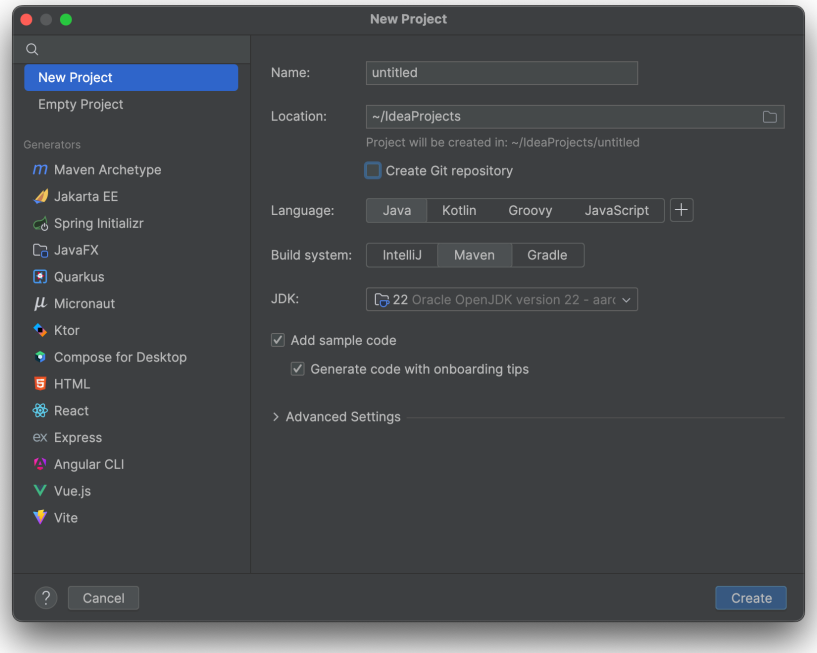
JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

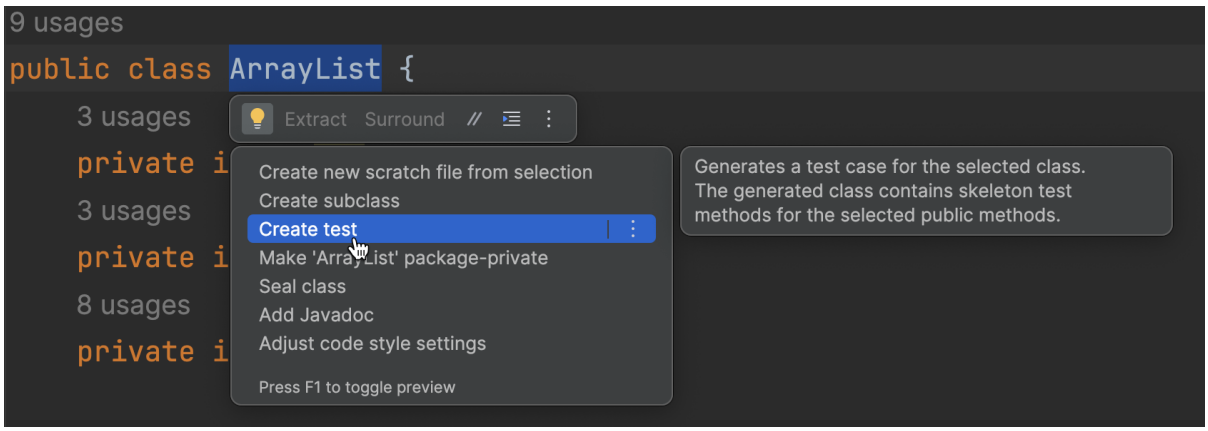
A unit test targets some other "class under test;" for example, the class `ArrayIntListTest` might be targeting the `ArrayIntList` as its class under test. A unit test generally consists of various testing methods that each interact with the class under test in some specific way to make sure it works as expected. JUnit isn't part of the standard Java class libraries.

## Creating a JUnit Test Case in IntelliJ

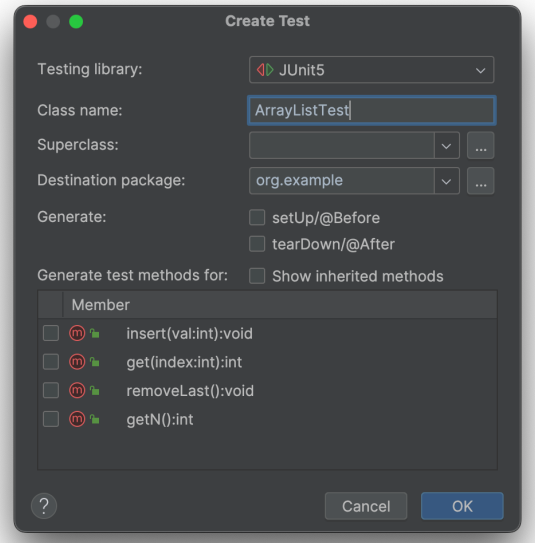
- Launch IntelliJ IDEA and click on "Create New Project."
- Choose a project type (e.g., Maven) and give your project a name.
- Click "Finish."



Right-click on the class for which you want to create a test class. Then, select **Show Context Actions** and choose **Create Test**. As shown below:




You will see a set of checkboxes to indicate which methods you want to test. IntelliJ will help you by creating test methods that you can fill in. (You can always add more later manually.) Choose the methods to test and click Ok.




When you're done, you should have a nice new JUnit test case file. I suggest that you change the `import` statement at the top to say the following:

```
import org.junit.*;
```

## Writing Tests

Each unit test method in your JUnit test case file should test a particular small aspect of the behavior of the "class under test." For example, an ArrayIntListTest might have one testing method to see whether elements can be added to the list and then retrieved. Another test might check to make sure that the list's size is correct after various manipulations. And so on. Each testing method should be short and should test only one specific aspect of the class under test.

JUnit testing methods utilize *assertions*, which are statements that check whether a given condition is true or false. If the condition is false, the test method fails. If all assertions' conditions in the test method are true, the test method passes. You use assertions to state things that you expect to always be true, such as `assertEquals(3, list.size());` if you expect the array list to contain exactly 3 elements at that point in the code. JUnit provides the following assertion methods:

| method name / parameters | description |
|---|---|
| **fail(String)** | Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented. |
| **assertTrue(true) / assertTrue(false)** | Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented. |
| **assertTrue([message], boolean condition)** | Checks that the boolean condition is true. |
| **assertsEquals([String message], expected, actual)** | Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays. |
| **assertEquals([String message], expected, actual, tolerance)** | Test that float or double values match. The tolerance is the number of decimals which must be the same. tolerance is the value that the 2 numbers can be off by. So it will assert to true as long as `Math.abs(expected - actual) < tolerance` |
| **assertNull([message], object)** | Checks that the object is null. |
| **assertNotNull([message], object)** | Checks that the object is not null. |
| **assertSame([String], expected, actual)** | Checks that both variables refer to the same object. |
| **assertNotSame([String], expected, actual)** | Checks that both variables refer to different objects. |

Here is a quick example that uses several of these assertion methods.

```
ArrayIntList list = new ArrayIntList();
list.add(42);
list.add(-3);
list.add(17);
list.add(99);

assertEquals(4, list.size());
assertEquals(17, list.get(2));
assertTrue(list.contains(-3));
assertFalse(list.isEmpty());
```

Notice that when using comparisons like `assertEquals`, expected values are written as the left (first) argument, and the actual calls to the list should be written on the right (second argument). This is so that if a test fails, JUnit will give the right error message such as, "expected 4 but found 0".

A well-written test method chooses the various assertion method that is most appropriate for each check. Using the most appropriate assertion method helps JUnit provide better error messages when a test case fails. The previous assertions could have been written in the following way, but it would be poorer style:

```
// This code uses bad style.
assertTrue(list.size() == 4);         // bad; use assertEquals
assertTrue(list.get(2) == 17);        // bad; use assertEquals
if (!list.contains(-3)) {
    fail();                           // bad; use assertTrue
}
assertTrue(!list.isEmpty());          // bad; use assertFalse and delete the !
```

Good test methods are short and test only one specific aspect of the class under test. The above example code is in that sense a poor example; one should not test `size`, `get`, `contains`, and `isEmpty` all in one method. A better (incomplete) set of tests might be more like the following:

```
@Test
public void testAddAndGet1() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    list.add(17);
    list.add(99);
    assertEquals(42, list.get(0));
    assertEquals(-3, list.get(1));
    assertEquals(17, list.get(2));
    assertEquals(99, list.get(3));

    assertEquals("second attempt", 42, list.get(0));   // make sure I can get them a second
time
    assertEquals("second attempt", 99, list.get(3));
}

@Test
public void testSize1() {
    ArrayIntList list = new ArrayIntList();
    assertEquals(0, list.size());
    list.add(42);
    assertEquals(1, list.size());
    list.add(-3);
    assertEquals(2, list.size());
    list.add(17);
    assertEquals(3, list.size());
    list.add(99);
    assertEquals(4, list.size());
    assertEquals("second attempt", 4, list.size());   // make sure I can get it a second
time
}

@Test
```

```
public void testIsEmpty1() {
    ArrayIntList list = new ArrayIntList();
    assertTrue(list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
}

@Test
public void testIsEmpty2() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
    list.remove(1);
    list.remove(0);
    assertTrue("after removing all elements", list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
}

...
```

You might think that writing unit tests is not useful. After all, we can just look at the code of methods like `add` or `isEmpty` to see whether they work. But it's easy to have bugs, and JUnit will catch them better than our own eyes.

Even if we already know that the code works, unit testing can still prove useful. Sometimes we introduce a bug when adding new features or changing existing code; something that used to work is now broken. This is called a *regression*. If we have JUnit tests over the old code, we can make sure that they still pass and avoid costly regressions.

## Annotations

The following table gives an overview of the available annotations in JUnit 5.

| Annotation | Description |
|---|---|
| **@Test public void method()** | The annotation @Test identifies that a method is a test method. |
| **@BeforeEach public void method()** | Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class). |
| **@AfterEach public void method()** | Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults). |
| **@BeforeAll public void method()** | Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database. |
| **@AfterAll public void method()** | Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database. |

## Running Your Test Case

a. Run Individual Test:
   - Click next to the test method name in the gutter (the vertical bar on the right side of the code editor).
   - Select "Run [testMethodName]" from the menu.

b. Run All Tests in a Class:
   - Click next to the test class name in the Project pane.
   - Select "Run [testClassName]" from the menu.

c. Test Results:
   - The "Run" tool window (usually at the bottom) will display the test results.
   - Green checkmarks indicate successful tests, while red X's indicate failures.